Available online at www.sciencedirect.com

ScienceDirect

Journal of Computer and System Sciences 73 (2007) 156–170

JOURNAL OF COMPUTER AND SYSTEM SCIENCES

www.elsevier.com/locate/jcss

Dispatch sequences for embedded control models ☆

Rajeev Alur *, Arun Chandrashekharapuram

Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104-6389, USA

Received 9 June 2005; received in revised form 8 December 2005

Available online 30 May 2006

Abstract

We consider the problem of mapping a set of control components to an executable implementation. The standard approach to this problem involves mapping control blocks to periodic tasks, and then generating a schedule. This schedule is platform-dependent, and its execution requires real-time operating system support. We propose an alternative approach which involves generating a dispatch sequence of control blocks in a platform-independent manner. Our solution relies on assigning *relative complexity* and *relative importance* measures to control components, and is an adaptation of the classical scheduling algorithms such as earliest–deadline–first. We show the benefits of our approach using simulation experiments on two case studies.

Keywords: Embedded software; Real-time scheduling; Model-based design

1. Introduction

Contemporary industrial control design already relies heavily on tools such as Simulink for mathematical modeling and simulation. Even though many such tools support implementation via automatic code generation from the model, many issues relevant to correctness and optimality of the implementation with respect to the timed semantics of the model are not satisfactorily addressed, and is tailored to a specific platform. Consequently, analysis results established for the model are not meaningful for the implementation and the code cannot be ported across platforms posing challenges for system integration. These challenges motivate our research.

In this paper, we focus on generating an executable implementation from a set B of control blocks. A control block computes outputs that influence other blocks or the environment being controlled. The control model has a well-defined timed semantics (either continuous or discrete) that can be used for simulation and analysis. Typically, the implementation relies on the support offered by a real-time operating system for scheduling periodic tasks. Each control block B_i is compiled into an executable code in a host language such as C, and the control designer specifies a period ρ_i for the corresponding task. To implement the resulting periodic tasks on a specific platform, one needs

☆ A preliminary version of this paper appears in Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2005, pp. 508–518. This research was partially supported by the US National Science Foundation under awards ITR/SY 0121431 and CCR-0410662.

* Corresponding author.
E-mail addresses: alur@cis.upenn.edu (R. Alur), arunc@cis.upenn.edu (A. Chandrashekharapuram).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2006.04.003

to determine the worst-case-execution-time τ_i for each block B_i, and check whether the task set is schedulable using standard scheduling algorithms such as earliest–deadline–first (EDF) or rate monotonic scheduling (cf. [4,17]).

While the real-time scheduling based implementation offers a separation of concerns using the abstraction of real-time tasks with periods and deadlines, it can hinder portability of control designs across platforms. As a concrete example, consider vision-based navigation of an autonomous robot trying to reach a target in a room full of obstacles. One control block computes the estimates of the obstacles while the other decides the trajectory based on the current estimates. Mapping these blocks to two tasks with specific periods introduces an abstraction that is not relevant to the high-level model or its goals. There are no *hard* real-time requirements in this application, and the performance can be measured by the time taken by the robot to reach the target. If the WCET (worst-case-execution-time) analysis on a particular processor reveals that the tasks are not schedulable, then in fact, the periods should be increased. If the analysis says that the tasks are schedulable, then it produces a schedule, which is a mapping from time slots to the tasks. This schedule is platform-dependent as it depends on the platform-specific WCET estimates. Moreover, executing the schedule requires real-time support from the operating system while the current trend in many application domains such as robotics is to employ commonly available computing platforms such as .NET [6]. Furthermore, since the scheduler views the tasks as periodic, it may leave the processor idle, thereby preventing improved performance.

In the proposed solution, our goal is to produce a *dispatch sequence* of blocks, rather than periodic tasks. The dispatch sequence is simply a string of control blocks, and is platform-independent. Unlike a schedule, a dispatch sequence has no notion of time slots or other real-time requirements. Ideally, we would like the sequence to be such that, on any given platform, it follows the reference trajectory of the continuous time model as best as one can on that platform. This goal is hard to quantify abstractly, and even if one could find a concrete measure for specific applications (for instance, the total distance traveled in the above robot example), we are not aware of any methods to generate sequences that optimize this measure in an efficient way. In this paper, we formulate the sequence generation problem, and propose a possible solution. We associate with each control block B_i a measure τ_i^r of *relative complexity* and a measure ρ_i^r of *relative importance*. The τ_i^r value is supposed to capture the computation time of B_i relative to the other blocks, and the ρ_i^r value is supposed to capture in a relative manner, how updating the output of B_i impacts the environment. We use the appropriately tightly scaled versions of ρ^r values as periods and of τ^r values as WCET estimates to generate sequences of blocks using the classical real-time scheduling algorithms such as non-preemptive EDF and EDF. Since EDF is preemptive and we want to generate an executable sequence of blocks, this step requires model transformation via block-code-splitting. The output of our strategy is a platform-independent and untimed sequence of blocks: executing this sequence does not require preemption or any support from real-time scheduler, and its ability to follow the reference trajectory on a particular platform depends on the processing power of the platform.

The rest of the paper is organized as follows. Section 2 describes our model for control blocks along with a continuous time and a discrete time semantics for the same. Section 3 describes the classical real-time scheduling based approach by formalizing schedules, schedule semantics and strategies for generating schedules using periods and WCET estimates. Section 4 defines the notion of a dispatch sequence, the associated semantics, and proposes strategies for generating dispatch sequences inspired by scheduling techniques, but using the notions of relative complexity and relative importance. Section 5 describes simulation experiments on two examples, one for robot navigation, and one for controlling heaters across multiple rooms, demonstrating the benefits of the proposed approach. We conclude with directions for future research in Section 6.

1.1. Related work

Bridging the gap between high-level modeling or programming abstractions, and implementation platforms has been identified as a key challenge for embedded software research by many researchers (cf. [18,19]). Programming abstractions for embedded real-time controllers include synchronous reactive programming (languages such as ESTEREL and LUSTRE [3,9,10]), and the related *Fixed Logical Execution Time* (FLET) assumption used in the Giotto project [12,13]. While these provide schedule-independent semantics, they do not address the problem of mapping continuous time controllers to an executable implementation. Recently, the problem of generating code from timed and hybrid automata has been considered in [1,14,21], but in these papers the focus has been on choosing the sampling period so as to avoid errors due to switching and communication. The work on mapping Simulink blocks to Lustre focuses on signal dependencies [5]. Model-based development of embedded systems is also promoted by other projects with orthogonal concerns: Ptolemy supports integration of heterogeneous models of computation [7] and

GME supports integration of multiple views of the system [16]. There is a rich literature on sampled control systems with a focus on understanding the gap between continuous and discrete controllers, determining the correct sampling period, and compensating for the computation delays in the design of control laws (cf. [2]). In scheduling literature, while many variations of the basic periodic scheduling problem have been explored, the focus is on determining a platform-dependent mapping from time slots to tasks. The most relevant of these is control-aware scheduling [20], where periods for tasks are determined by optimizing a performance index.

2. Modeling controllers

In this section, we describe the model of a real-time control system and the desired semantics for the model.

2.1. Model

Let X be a finite set of *environment variables* modeling the physical world to be controlled, and U be a finite set of *control variables* to be computed by the control software. Each variable has a type, which typically is $\mathbb{R}$, the set of reals. A state over a set W of variables is a mapping from W to values. We use Q_W to denote the set of all states over W. A control model is given by $M = \langle M_C, M_E \rangle$, where M_C is the *controller model* and M_E is the *environment model*.

The controller model M_C consists of a finite set B of *control blocks*, where each control block B_i in B has the following components:

- A set of input variables $Y_i \subseteq (X \cup U)$, which the block reads to do its computation.
- A set of output variables $U_i \subseteq U$, which the block writes after its computation.
- A relation $f_i \subseteq Q_{Y_i} \times Q_{U_i}$, defining the computation of the block.
- A set of initial states $Q_i^0 \subseteq Q_{U_i}$ for the output variables of the block.

The following properties must be satisfied by M_C:

- Every output variable must be computed by a unique block. That is, for all i, j with $i \neq j$, $U_i \cap U_j$ must be empty, and $\bigcup_j U_j$ must equal U.
- Consider a directed graph B_G whose nodes are control blocks and where there is an edge from B_i to B_j if B_j reads an output variable computed by B_i. Then B_G must be acyclic.

The environment model M_E is given by

- A relation $g_x \subseteq Q_X \times Q_U \times \mathbb{R}$ for every environment variable $x \in X$. This relation is used to define the rate of change of x in terms of the current state.
- A set of initial states $Q^0 \subseteq Q_X$ for the environment variables.

We have allowed our models to be non-deterministic, but this choice is not central to this paper, and in many cases, the computation of each control block B_i is defined by a function $f_i : Q_{Y_i} \to Q_{U_i}$, and the rate of change of an environment variable x is given by a function $g_x : Q_X \times Q_U \to \mathbb{R}$.

2.2. Robot navigation example

Consider a robot R which can move on a 2-D plane (see Fig. 1). Initially R is at the (fixed) starting point S. Its goal is to reach a (fixed) target point T, without colliding with any of the stationary circular obstacle-disks O_1, O_2 and O_3 on the plane. The robot moves in the direction θ at a constant speed v_R. It can estimate the obstacles only approximately, and we assume that the estimate is a circle whose center coincides with the center of the obstacle (x_c, y_c) and whose radius r is always larger than the actual radius r_0. The estimation rule is given by

$$r = r_0 + \left(\sqrt{(x_c - x)^2 + (y_c - y)^2} - r_0\right)^2 / 500,$$

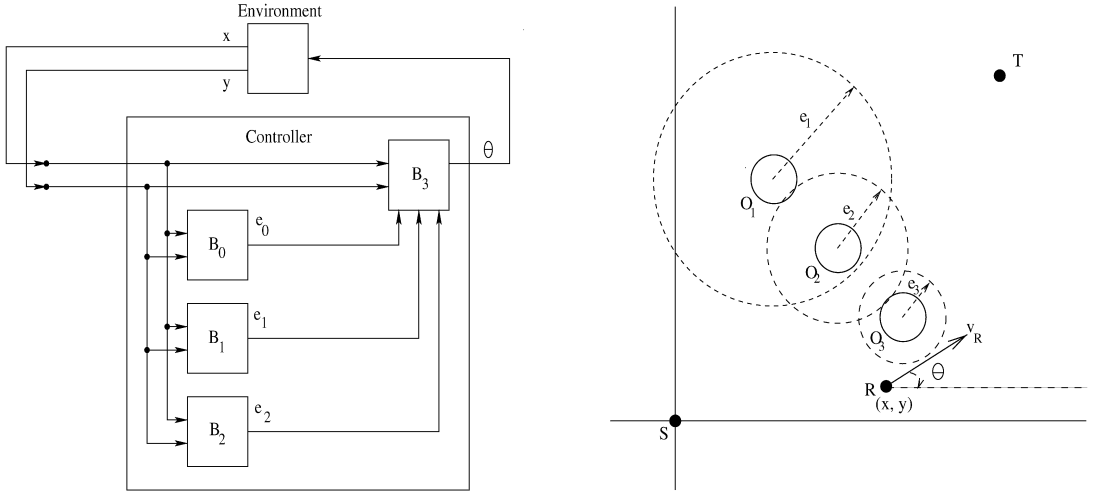

Fig. 1. Robot navigation example.

where (x, y) is the current position of R. The estimate r is smaller if R is closer to the obstacle. Based on the estimated radii of the 3 obstacles from the current position, the robot computes θ as follows: first, it checks if the direct path from the current position (x, y) to the target T faces no obstruction—if so, it proceeds in that direction. If not, it computes the slopes of the tangents from the current position to the estimated obstacle circles, and checks whether rays along the tangents face any obstruction. Then, among the rays without any obstruction, it chooses to go along that ray which makes the least angle with the direct path to the target. Figure 1 shows a snapshot of the robot position during its motion, along with the estimated obstacle radii and the selected direction of motion.

Figure 1 also shows a block diagram of the model. The environment variables are the coordinates (x, y) of the robot position. The initial values of (x, y) are the coordinates of S. The differential equations governing the rates of change of x and y are:

$$\dot{x} = v_R \cos\theta, \qquad \dot{y} = v_R \sin\theta.$$

The control variables are e_0, e_1, e_2, and θ, where e_i is the estimate of the radius of obstacle O_i. There are four control blocks $B_0, \ldots, B_3$. The control block B_i, for $0 \leqslant i \leqslant 2$, is used to estimate radius of the obstacle O_i. Its input variables are x and y, and its output variable is e_i. The control block B_3 is used to calculate θ. Its input variables are e_0, e_1, e_2, x, and y, and its output variable is θ. The initial values of e_i are the estimates from S, and that of θ is the angle computed using the initial values of e_i.

2.3. Semantics

Given a model M over variables X and U, a *trajectory* for M is a function $\psi : \mathbb{R} \to Q_{X \cup U}$. A *semantics* for a model M, denoted $[\![M]\!]$, is a set of trajectories for M. Two semantics, *continuous time* and *parameterized discrete time*, are described below.

2.3.1. Continuous time semantics

The continuous time semantics for a model M, denoted $[\![M]\!]_C$, evaluates all control variables at *every* point in the continuous time domain. It consists of the trajectories ψ satisfying the following constraints: for all $t \in \mathbb{R}$, $t \geqslant 0$, and for all $B_i \in B$ we have

$$\psi(0)(X) \in Q^0, \quad \big(\psi(t)(X), \psi(t)(U), \dot{\psi}(t)(X)\big) \in g_x; \tag{1}$$

and

$$\psi(0)(U_i) \in Q_i^0, \quad \big(\psi(t)(Y_i), \psi(t)(U_i)\big) \in f_i. \tag{2}$$

2.3.2. Parameterized discrete time semantics

The parameterized discrete time semantics for a model M evaluates the *control variables* with a sampling period of Δ, and a zero-order hold. So, all control outputs are piecewise-constant, the pieces being of length Δ.

Let $t_k = k\Delta$ for $k \in \mathbb{N}$. Given a $\Delta > 0$, the discrete time semantics for M, denoted by $[\![M]\!]_D^{\Delta}$, is a set of trajectories ψ satisfying the following constraints, besides (1): for all $B_i \in B$ and for all $k \in \mathbb{N}$,

$$\psi(0)(U_i) \in Q_i^0, \quad \big(\psi(t_k)(Y_i), \psi(t_k)(U_i)\big) \in f_i,$$

and for $t_{k-1} \leqslant t < t_k$, $t \in \mathbb{R}$,

$$\psi(t)(U_i) = \psi(t_{k-1})(U_i).$$

We note that the continuous time semantics is the ideal semantics for any given model. The discrete time semantics introduces an error into the model because of the zero-order hold for Δ intervals. We may want to define the error using some metric over trajectories, but it is difficult to quantify the errors abstractly. For specific applications, such as those evaluated in this paper, we can find some concrete measures to quantify the performance of a trajectory, and use them to compare any two trajectories. In our robot navigation example, total distance traveled from the start position to the target is a reasonable measure of performance.

3. Schedule-based implementation

In this section, we discuss some standard implementation strategies to generate real-time tasks from a given model $M = \langle M_C, M_E \rangle$. We first define the notion of a *schedule* and then discuss the standard platform-dependent ways of computing schedules.

We assume henceforth that the minimum time unit of execution of a control task is 1. That is, the values of the control variables can be *updated* by any control function only in intervals of one time unit. This simplifies the notation, otherwise we would need definitions parameterized by Δ as in case of discrete-time semantics.

3.1. Schedule and schedule semantics

A schedule is a mapping from time slots to blocks, which indicates the block that executes in each time slot. The schedule semantics for a schedule is the set of trajectories obtained by executing the blocks according to the schedule: an instantiation of a block executes only in the time slots given by the schedule; its input values are read at the beginning of the first time slot of its execution and the control outputs computed by the block are updated at the end of the last slot of its execution. This type of predictable execution can be implemented using the time-triggered architecture [17].

Formally, a *schedule* sch for M is a function $\mathrm{sch} : \mathbb{N} \to B \cup B^+ \cup \{\perp\}$, where $B^+ = \{B_i^+ \mid B_i \in B\}$ is used to denote the completion of the current instances of the corresponding tasks, and $\perp$ denotes *idle*. The connotation of a schedule is as follows. Let *slot* k denote the time interval $[k-1, k]$. Then for $k \geqslant 1$,

$$\mathrm{sch}(k) = \begin{cases} B_i \text{ means } B_i \text{ executes in slot } k \text{ but } f_i \text{ is not yet computed.} \\ B_i^+ \text{ means } B_i \text{ executes and finishes computation of } f_i \text{ in slot } k. \\ \perp \text{ means the processor is idle in slot } k. \end{cases}$$

Given a schedule sch, the semantics associated with the model M, denoted by $[\![M]\!]_{\mathrm{sch}}$, is a set of trajectories obtained by executing the blocks according to sch. For example, consider the schedule $B_0 B_1 B_1 B_0^+ B_1^+ \ldots$. Block B_0 starts executing at time $t = 0$ after reading its inputs and executes in time slot 1. It is then preempted at time $t = 1$ when B_1 starts executing. Block B_0 again executes in time slot 4, and finishes its execution in that time slot. The values computed by B_0 are updated at the end of slot 4. We assume that reading and updating take zero computation time. Therefore, $\psi(t)(U_0) = q_0$ for some $q_0 \in Q_0^0$ for $0 \leqslant t < 4$, and $\psi(4)(U_0) = f_0(\psi(0)(Y_0))$.

Formally, $[\![M]\!]_{\mathrm{sch}}$ consists of the trajectories ψ satisfying the following constraints, besides (1): for all $k \in \mathbb{N}$ and all $B_i \in B$,

$$\psi(0)(U_i) \in Q_i^0,$$
$$\psi(t)(U_i) = \psi(k-1)(U_i) \quad \text{for } (k-1) < t < k,$$

$$\psi(k)(U_i) \in f_i\big(\psi(l)(Y_i)\big) \quad \text{if } \mathrm{sch}(k) = B_i^+, \text{ where } l \text{ is the smallest } l' \text{ such that}$$
$$\mathrm{sch}(l') = B_i \text{ and } \forall j\colon l' < j < k\colon \mathrm{sch}(j) \neq B_i^+;$$
$$l = k \text{ if no such index } l' \text{ exists,}$$
$$\psi(k)(U_i) = \psi(k-1)(U_i) \quad \text{otherwise.}$$

3.2. Algorithms for computing schedules

Given a model M, the following steps are typically followed:

(1) We first generate one task T_i for each block B_i in the model. The code executed by the task will be the function f_i, and the values used as input for variables in Y_i will be the most recently computed values for those variables.
(2) We then assign a period $\rho(B_i)$, where $\rho : B \to \mathbb{N}$, to each task T_i. The period $\rho(B_i)$, also denoted by ρ_i, is *independent of the platform* on which the tasks are going to be executed. That is, as long as the task set is schedulable, the periods remain the same. They are usually assigned by control engineers to satisfy the performance requirements of the control model such as stability, ability to track a given trajectory, etc. (cf. [2]). The relative deadline of T_i is equal to ρ_i, and this means that the task must be executed once every period.
(3) Then, given an execution platform F, we compute $\tau : B \to \mathbb{N}$, where $\tau(B_i)$, also denoted as τ_i, is the Worst-Case-Execution-Time (WCET) of B_i on F. The WCETs can be estimated using well-known WCET estimation methods (cf. [11]).
(4) Given ρ and τ, we can execute the tasks using a real-time operating system (RTOS) that includes a real-time scheduler for periodic tasks.

The RTOS typically uses well-known hard real-time scheduling algorithms for executing the tasks. We use two scheduling algorithms in this paper: the earliest–deadline–first (EDF) algorithm and the non-preemptive earliest–deadline–first (NPEDF) algorithm. The EDF (cf. [4]) algorithm is a preemptive algorithm. When a new task is released or when the current task completes execution, it schedules the task with the earliest deadline among all active tasks. The NPEDF algorithm (cf. [15]) schedules the task with the earliest deadline among all active tasks, if the processor is idle or the currently executing task has finished execution.

For a given ρ and τ, if the task set is schedulable by EDF, it produces a periodic schedule sch, and the semantics $[\![M]\!]_{\mathrm{EDF}(\rho,\tau)}$ is defined to be $[\![M]\!]_{\mathrm{sch}}$. If the task set is not schedulable using EDF, then the semantics $[\![M]\!]_{\mathrm{EDF}(\rho,\tau)}$ is undefined. The semantics associated with the NPEDF algorithm $[\![M]\!]_{\mathrm{NPEDF}(\rho,\tau)}$ is defined in a similar way.

We call this approach *platform dependent* since the schedule depends on the *concrete* values of the WCET estimates τ. Note that the only feature of the platform relevant in our context is its processing power, which is captured by the WCET estimates τ.

Consider the robot navigation example again. For this model, four tasks would be generated: T_i, $0 \leqslant i \leqslant 2$, for estimating the radii of the obstacles, and T_3 for calculating θ based on the estimates. An assignment of periods for the tasks, and WCET estimates on three different platforms F_1, F_2 and F_3 is given in Fig. 2. Platform F_1 is the fastest while F_3 is the slowest. The tasks are schedulable by NPEDF (a schedulability test for NPEDF can be found in [15]) on F_1 and F_2 but not on F_3. For $t \in [1..120]$ (120 is the LCM of the periods of the tasks), the schedule produced by NPEDF on F_1 and F_2 is shown in Fig. 3. The schedule produced by EDF on F_2 is also shown. The notation $[i : t_1 - t_2]$

Task	ρ (ms)	τ (ms)		
		F_1	F_2	F_3
T_0	120	12	24	28
T_1	120	12	24	28
T_2	120	12	24	28
T_3	24	3	6	7

Fig. 2. Sample periods and execution times for robot navigation example.

Strategy	Platform	Schedule in [1..120]
NPEDF	F_1	$[3:1-3^+]$ $[0:4-15^+]$ $[1:16-27^+]$ $[3:28-30^+]$ $[2:31-42^+]$ $[\perp:43-48]$ $[3:49-51^+]$ $[\perp:52-72]$ $[3:73-75^+]$ $[\perp:76-96]$ $[3:97-99^+]$ $[\perp:100-120]$
NPEDF	F_2	$[3:1-6^+]$ $[0:7-30^+]$ $[3:31-36^+]$ $[1:37-60^+]$ $[3:61-66^+]$ $[2:67-90^+]$ $[3:90-96^+]$ $[3:97-102^+]$ $[\perp:103-120]$
EDF	F_2	$[3:1-6^+]$ $[0:7-24]$ $[3:25-30^+]$ $[0:31-36^+]$ $[1:37-48]$ $[3:49-54^+]$ $[1:55-66^+]$ $[2:67-72]$ $[3:73-78^+]$ $[2:79-96^+]$ $[3:97-102^+]$ $[\perp:103-120]$

Fig. 3. Schedules generated by NPEDF and EDF.

means that block B_i executes continuously from time slot t_1 to time slot t_2 but without completing its execution, and $[i:t_1-t_2^+]$ means that B_i executes continuously from time slot t_1 to time slot t_2 and completes its execution at t_2.

We first note here that the periods (and therefore deadlines) assigned to the tasks are artificial. For example, if a task set is not schedulable, the control engineer might be able to increase the periods without violating the performance requirements of the control model. Here, we can increase the periods slightly to render the tasks schedulable on F_3. Further, we observe that there are a lot of idle times on F_1, whereas executing the control tasks without any idle times (that is, executing the next block in sequence immediately after a block finishes execution) can improve performance. The goal in this case is to approximate the discrete semantics $[\![M]\!]_D^1$ (and hence the continuous semantics $[\![M]\!]_C$) as best as possible given the processing constraints. Abstracting this goal to scheduling of the tasks with deadlines and periods loses too much information. The performance measure in this case is the total distance traveled, or equivalently, time to reach the target, and we would like a systematic and computationally tractable approach which will minimize this performance measure.

4. Dispatch sequences

In this section, we discuss our method of implementing controllers without real-time tasks. We introduce the notion of a *dispatch sequence* which is a string of blocks indicating the order in which blocks are to be executed. Then, after defining the semantics associated with dispatch sequences, we describe strategies to generate them using NPEDF and EDF.

4.1. Dispatch sequence semantics

A *dispatch sequence* $\sigma \in B^\star$ is a string over B which indicates the *sequence* in which the blocks should be executed repeatedly. The whole block is to be executed without preemption, and when it completes its execution, the succeeding block can start executing immediately. *Unlike a schedule, there is no notion of time in a dispatch sequence*. Hence, dispatch sequences may look like cyclic executive schedules, but are different.

Given a platform F, let $\gamma_l, \gamma_u : B \to \mathbb{N}$ be two functions that specify lower and upper bounds respectively on the execution time of B_i on F. That is τ_i, the execution time of an instance of B_i on F, is such that $\gamma_l(B_i) \leqslant \tau_i \leqslant \gamma_u(B_i)$. Note that different executions of the same block can take different amounts of time, and nothing is said about the distribution of τ_i between the two limits.

Given a triple $(\sigma, \gamma_l, \gamma_u)$, the dispatch-sequence semantics associated with a model M, denoted by $[\![M]\!]_{(\sigma,\gamma_l,\gamma_u)}$, is the set of trajectories obtained by executing the blocks according to σ, where the execution times for the blocks are chosen according to the bounds. Formally, it can be defined as follows.

Let $|\sigma| = k$, and let σ_i denote the ith block in σ for $i \geqslant 1$. Define $\mathrm{Sch}(\sigma, \gamma_l, \gamma_u)$, to be the set of all schedules $\mathrm{sch} : \mathbb{N} \to B \cup B^+$ such that there exists a sequence $i_0 = 0 \leqslant i_1 \leqslant i_2 \leqslant \cdots$ for which for all $j \geqslant 1$, if $m = j \bmod k$, then

$$\gamma_l(\sigma_m) \leqslant (i_j - i_{j-1}) \leqslant \gamma_u(\sigma_m) \quad \text{and}$$
$$\mathrm{sch}(n) = \begin{cases} \sigma_m & \text{for } (i_{j-1}+1) \leqslant n < i_j, \\ \sigma_m^+ & \text{for } n = i_j. \end{cases}$$

The semantics $[\![M]\!]_{(\sigma,\gamma_l,\gamma_u)}$ is defined to be the union $\bigcup_{\mathrm{sch}\in\mathrm{Sch}(\sigma,\gamma_l,\gamma_u)} [\![M]\!]_{\mathrm{sch}}$.

Block	γ_l (ms)	γ_u (ms)
B_0	22	24
B_1	22	24
B_2	22	24
B_3	4	6

Fig. 4. γ_l and γ_u for the blocks in robot navigation for platform F_2.

Block	τ_i^r	ρ_i^r
B_0	4	5
B_1	4	5
B_2	4	5
B_3	1	1

Fig. 5. Relative execution times and relative periods for blocks of robot navigation.

For example, consider the *round-robin* (RR) dispatch-sequence $\sigma^{\mathrm{RR}} = (B_0B_1B_2B_3)^*$ for the navigation example. The blocks are to be executed repeatedly in the order $B_0B_1B_2B_3$. Figure 4 gives the γ_l and γ_u values for the platform F_2. This means that B_i for $i = 0, 1, 2$ can execute for anytime between 22 and 24 ms, and B_3 for anytime between 4 and 6 ms. Here, estimation takes much longer than computing the direction, and round-robin does not seem to be a desirable choice.

4.2. Relative execution times and relative periods

Since we do not want to commit to concrete deadlines and periods, we introduce the notion of "relative" periods and "relative" execution times. Let a controller model M_C with n blocks be given. For each block B_i, we assign a *relative execution time* $\tau_i^r \in \mathbb{N}$ and a *relative period* $\rho_i^r \in \mathbb{N}$ such that $\gcd(\tau_1^r, \tau_2^r, \ldots, \tau_n^r) = \gcd(\rho_1^r, \rho_2^r, \ldots, \rho_n^r) = 1$. The relative execution time τ_i^r is an estimate of the WCET of B_i on *any* platform, relative to the times taken by other blocks in the model. We can compute them by several approximate methods. One method is to scale the execution times of B_i on several platforms by the speeds of those platforms, and take the average of the scaled times as the estimate of τ_i^r. The ratio of the WCETs of two blocks can be different on different platforms due to factors such as cache size and floating-point processing units. The assignment of relative execution times assumes a uniform ratio, and this implies that we need to be conservative in the estimates of WCETs. The relative period ρ_i^r is an index of the importance assigned to the block, when compared to the importance of other tasks. These are to be assigned by the control engineer.

Figure 5 shows a set of relative execution times and relative periods for the blocks of the robot navigation example. Note that the WCETs of the blocks on the platform F_2 as given in Fig. 4 are roughly 6 times the relative execution times as given by Fig. 5. In general, $\gamma_l(B_i)$ and $\gamma_u(B_i)$ are expected to be roughly k times τ_i^r for some scaling factor k.

The dispatch-sequence generation problem can be stated informally as follows. Given a model M and relative measures τ^r and ρ^r, generate a string σ of blocks such that, on any platform F where the lower and upper bounds γ_l and γ_u for blocks are consistent with the ratios given by τ^r, the trajectories in $[\![M]\!]_{(\sigma,\gamma_l,\gamma_u)}$ are as close as possible to the trajectories in $[\![M]\!]_C$. There does not seem to be a computationally tractable way of formulating this as a mathematical optimization problem. Hence, we settle for heuristics inspired by the classical scheduling schemes.

4.3. Dispatch sequence generation using NPEDF

In this section, we explain our strategy to generate dispatch sequences using NPEDF algorithm from given model M_C and relative measures. The dispatch sequence, denoted by σ^{NPEDF}, is such that a block is always executed in its entirety.

The main steps to generate σ^{NPEDF} are as follows:

(1) Compute the relative utilization $U^r = \sum_{i=1}^{n} \tau_i^r/\rho_i^r$ of the blocks. If $U^r > 1$, then scale the periods ρ_i^r by the smallest integer p such that $U^r/p \leqslant 1$; otherwise, let $p = 1$. Call these new periods, the scaled versions of ρ_i^r.
(2) Compute $l = \mathrm{lcm}(p\rho_1^r, p\rho_2^r, \ldots, p\rho_n^r)$. This is the lcm of the scaled periods.
(3) Run the NPEDF algorithm from time $t = 0$ to time $t = l$ with τ_i^r as the execution time and $p\rho_i^r$ as the period of each task B_i to get a schedule sch(NPEDF) of length l. Since $U^r \leqslant 1$, all the instances of the blocks released before $t = l$ are executed before $t = l$.
(4) In sch(NPEDF), there may be some idle times. Collapse the schedule by disregarding the idle times to obtain a dispatch sequence σ' from sch(NPEDF). That is, if there is any idle time between two successive blocks B_i and B_j in sch(NPEDF) then B_j follows immediately after B_i in σ', and the idle time after the execution of the last

block in sch(NPEDF) is discarded. The desired dispatch sequence σ^{NPEDF} is σ'. It is easy to see that σ^{NPEDF} as obtained above is indeed a string over B.

For example, consider the relative execution times and periods for the robot navigation example given in Fig. 5. The relative utilization U^r is 17/5. We scale this by $p = 4$. We then obtain $l = \text{lcm}(4, 20, 20, 20) = 20$. We then simulate it using NPEDF algorithm from $t = 0$ to $t = 20$ to get the schedule

$$[3 : 1 - 1^+]\ [0 : 2 - 5^+]\ [3 : 6 - 6^+]\ [1 : 7 - 10^+]\ [3 : 11 - 11^+]$$
$$[2 : 12 - 15^+]\ [3 : 16 - 16^+]\ [3 : 17 - 17^+]\ [\perp : 18 - 20].$$

We then get σ' from the above schedule by removing idle times: there are three slots of idle time at the end for this schedule, and so σ^{NPEDF} is

$$(B_3\ B_0\ B_3\ B_1\ B_3\ B_2\ B_3\ B_3).$$

A property of σ^{NPEDF} is that if the above algorithm was executed with $\alpha\tau_i^r$ for some $\alpha \in \mathbb{N}$ as the execution times instead of τ_i^r, then the dispatch sequence produced is the same irrespective of α. In other words, if the execution times are scaled by α and the periods by $p.\alpha$, where p is as in the above algorithm, and the tasks are scheduled using NPEDF, then the schedules obtained are the same as the schedules corresponding to the dispatch sequence σ^{NPEDF}. This means that the *dispatch-sequence generation algorithm* needs to be run only once, regardless of the platform on which the dispatch sequence is going to execute.

Theorem 1 *(Scaling theorem). Let M be a given model with relative execution times τ_i^r and relative periods ρ_i^r for each block B_i, and let σ^{NPEDF} be the corresponding dispatch sequence. Let $p \in \mathbb{N}$ be the least integer such that $\sum_i (\frac{\tau_i^r}{p.\rho_i^r}) \leqslant 1$. Given an $\alpha \in \mathbb{N}$, let $\tau, \rho : B \to \mathbb{N}$ such that $\tau(B_i) = \alpha.\tau_i^r$ and $\rho(B_i) = \alpha.p.\rho_i^r$. Then, for any schedule* sch*,* $\text{sch} \in \text{Sch}(\sigma^{\text{NPEDF}}, \tau, \tau)$ *iff* sch *is generated by* $\text{NPEDF}(\tau, \rho)$.

Proof. Call the set of assignments of execution times τ_i^r and periods $p.\rho_i^r$ the un-scaled version and the concrete set of assignments τ and ρ as the scaled version. Then it is enough to prove that

Claim 1. *For all $j \geqslant 1$, if the jth block scheduled by the NPEDF algorithm when run on the un-scaled version is the kth instance of B_i for some k and i, then the jth block scheduled by NPEDF when run on the scaled version is the kth instance of B_i. Moreover, if the jth block is scheduled at time t_j in the un-scaled version, then it is scheduled at time $t'_j = \alpha.t_j$ in the scaled version.*

Without loss of generality, let the periods ρ_i^r be in non-decreasing order. The proof is by induction on j.

Base case: For $j = 1$, the first block scheduled in un-scaled version is the first instance of B_1, since B_1 has the least period $p.\rho_1^r$ and hence the earliest deadline. It is scheduled at time $t_1 = 0$. In the scaled version, B_1 still has the least period namely $\alpha.p.\rho_1^r$, and hence its first instance is scheduled first by the algorithm. It is scheduled at time $t'_1 = 0 = \alpha.t_j$. Hence, the claim holds for $j = 1$.

Induction step: Let the claim hold for all $j < m$. We will prove that it holds for $j = m$.

Let the mth block scheduled by the algorithm at t_m in the un-scaled version be the kth instance of some block B_i. By the induction hypothesis, $t'_{m-1} = \alpha.t_{m-1}$, and the $(m-1)$th block scheduled is the same in both the un-scaled and the scaled versions. Since the computation times, and the periods are both scaled by α, the time elapsed from the end of execution of the $(m-1)$th block in the scaled version and t'_m is α times the corresponding time in the un-scaled version. Therefore, $t'_m = \alpha.t_m$.

(a) Now, the number of instances of any block B_l released at or before t_m is $\lfloor \frac{t_m}{p.\rho_l^r} \rfloor$. The number of instances of B_l released at or before t'_m is $\lfloor \frac{\alpha.t_m}{\alpha.p.\rho_l^r} \rfloor = \lfloor \frac{t_m}{p.\rho_l^r} \rfloor$. Thus, in both the unscaled and scaled versions, the same number of instances of every block is released. By the induction hypothesis, since the same instances of each block have been executed before t_m in the un-scaled version and before t'_m in the scaled version, the set of instances from which the next task to be scheduled at t_m and t'_m in the un-scaled and scaled versions is the same. Call this set ζ.

(b) Now, note that if an instance I in ζ is released at some time t_I in the un-scaled version, then it is released at time $\alpha.t_I$ in the scaled version. Therefore, if the deadline of I in the un-scaled version is $t_I + p.\rho_i^r$, then its deadline in the scaled version is $\alpha.(t_I + p.\rho_i^r)$ in the scaled version. Therefore, for any two instances I_1 and I_2 in ζ, if deadline$(I_1) <$ deadline(I_2) in the un-scaled version, then the same holds in the scaled version also.

Therefore, from the (a) and (b) above, the mth instance to be scheduled by both the un-scaled and the scaled versions is unique, namely, the instance with the earliest deadline among the blocks in ζ (assume that the algorithm has a deterministic choice when two instances have the same deadline, say, the instance with the lower index in the listing of blocks). □

4.4. Dispatch sequence generation using EDF

The dispatch-sequence generation algorithm using EDF is similar to the one using NPEDF, except that when we use EDF, the resulting sequence is no longer a string over B since some blocks might be preempted. In other words, the block-code of B_i (that is, the code implementing f_i) may need to be split. We first discuss how to handle splitting of block-code before proceeding to dispatch-sequence generation.

Given a block B_i, we assume that we can split the block-code of B_i into τ_i^r contiguous portions such that the relative execution times of each contiguous portion is approximately the same. We can then create τ_i^r blocks $B_{i1}, \ldots, B_{i\tau_i^r}$ such that B_{il} executes the lth contiguous portion, and $\tau_{il}^r = 1$ for all l. The inputs of B_{i1} are the inputs of B_i, and the inputs of B_{il} for $l > 1$ are the outputs of $B_{i(l-1)}$; the outputs of B_i are the outputs of $B_{i\tau_i^r}$. Call B_{ij} the *split-block* of B_i and the new model M' with $B' = \{B_{ij}\}$ as the set of blocks as the *split-model* of M. Note that M' is a semantics-preserving transformation of M.

The main steps to generate σ^{EDF} are as follows:

(1) Compute the utilization $U^r = \sum_{i=1}^{n} \frac{\tau_i^r}{\rho_i^r}$ of the blocks. If $U^r > 1$, then scale the periods ρ_i^r by the smallest integer p such that $U^r/p \leqslant 1$; otherwise, let $p = 1$. Call these new periods, the scaled versions of ρ_i^r.
(2) Compute $l = \text{lcm}(p\rho_1^r, p\rho_2^r, \ldots, p\rho_n^r)$. This is the lcm of the scaled periods.
(3) Produce the split-model M' of M.
(4) Run the EDF algorithm from time $t = 0$ to time $t = l$ with τ_i^r as the execution time, and $p.\rho_i^r$ as the period of task B_i to get a schedule sch(EDF) of length l. Since $U^r \leqslant 1$, all the instances of the blocks released before $t = l$ are executed before $t = l$. Now, the EDF algorithm can split the block B_i by preempting it. Thus, sch(EDF) is a mapping from $\mathbb{N}$ to $B \cup B^+ \cup \perp$, and it can be viewed as a mapping from $\mathbb{N}$ to $B' \cup \perp$, by replacing τ_i^r time slots allocated to an instance of B_i by the τ_i^r split blocks B_{ij}.
(5) In sch(EDF), there may be some idle times. Collapse the schedule by disregarding the idle times to obtain a dispatch sequence $\sigma' \in (B')^\star$ from sch(EDF). That is, if there is any idle time between two successive blocks B_{il} and B_{jk} in sch(EDF) then B_{jk} follows immediately after B_{il} in σ'; further, the idle time after the execution of the last block in sch(EDF) is discarded.
(6) Now, note that a block B_i need not be split by the EDF algorithm into τ_i^r split-blocks. In other words, $B_{i(l+1)}$ may *always* follow B_{il} in σ'. Therefore, we can optimize splitting of M by finding maximal sequences $B_{il}B_{i(l+1)}\ldots B_{i(l+j)}$ of split-blocks of B_i which always execute contiguously in σ', and combine all the blocks in a sequence into a single block. Let B'' be the new set of blocks obtained after performing this optimization step, and the final schedule σ^{EDF} is in $(B'')^\star$.

Note that all the steps above can be automated. As an illustration, consider again the robot navigation model, whose relative execution times and periods are given in Fig. 5. The utilization U^r is 17/5. We scale this by $p = 4$. We then obtain $l = \text{lcm}(4, 20, 20, 20) = 20$. We then simulate it using EDF algorithm from $t = 0$ to $t = 20$ to get the schedule

$$[3:1-1^+]\,[0:2-4]\,[3:5-5^+]\,[0:6-6^+]\,[1:7-8]\,[3:9-9^+]$$
$$[1:10-11^+]\,[2:12-12]\,[3:13-13^+]\,[2:14-16^+]\,[3:17-17^+]\,[\perp:18-20].$$

We then get $\sigma' \in B'$ from the above schedule by removing the three idle slots from the end of the schedule. We then obtain B'' from B' as follows: $B_3 \in B''$ since $\tau_1^r = 1$. Since B_0 is split into two parts whose relative execution times

are 3 and 1, respectively, the first three split-blocks of B_0 can be combined into a single block B'_{01}. Similarly, B_{11} and B_{12} can be combined into a single block B'_{11}, and B_{13} and B_{14} can be combined into B'_{12}. Again, last three blocks of B_2 can be combined into a single block B'_{22}. Thus, $B'' = \{B'_{01}, B_{03}, B'_{11}, B'_{12}, B_{21}, B'_{22}, B_3\}$. Therefore, σ' can be written as

$$\left(B_3\ B'_{01}\ B_3\ B_{03}\ B'_{11}\ B_3\ B'_{12}\ B_{21}\ B_3\ B'_{22}\ B_3\right)$$

to give the final dispatch sequence σ^{EDF}.

We note here that the *exact* splitting of block-codes to get B'' is non-trivial. However, since there are no hard real-time requirements, and the purpose of the intended strategy is to improve performance, there is no need for exact splitting. To ensure that splitting a block does not change its meaning, adequate information must be stored and retrieved across the splitting boundary. Implementing this split correctly is challenging, but this issue is beyond the scope of this paper.

5. Evaluation and experimental results

In this section, we evaluate the performance of dispatch sequences generated using NPEDF and EDF, against those of round-robin dispatch sequences and the schedule-based platform-dependent implementation strategies. We first describe the simulator used to perform our experiments. We then examine the results in the case of two case studies: the first being the robot navigation example used in the previous sections, and the second, a house-heater system. In our analysis, we focus on the impact of scheduling on the performance of the system assuming the scheduling overhead is negligible. Since our method computes the schedules statically, and the schedules are independent of the platform, it is clear that the scheduling overhead of our approach is less than the classical methods.

5.1. Simulator

The inputs to the simulator are the following:

- *Model* $M = \langle M_C, M_E \rangle$: An input file provides information about the structure of the model. It lists the environment variables, the control blocks $B_i \in B$ in the order given by topological sort of B_G and U_i, Y_i, τ_i^r, ρ_i^r, $\gamma_l(B_i)$, and $\gamma_u(B_i)$ for every B_i. The file also indicates the function to be used. Finally, the simulator needs initial values of all the variables.
- *Simulation step* δ: To approximate the continuous-time semantics, the simulator needs an integration step δ such that $0 \leqslant \delta < 1$.
- *Simulation time* N: It simulates from $t = 0$ to $t = N$.

The simulator simulates M as per the continuous time and parameterized discrete time semantics, as per the concrete NPEDF and EDF scheduling strategies as described in Section 3, and by using the dispatch sequences generated by the round-robin, NPEDF, and EDF strategies as described in Section 4. Each of these cases is briefly discussed below:

- *Continuous-time semantics* (`cont`): The simulation is carried out in steps of δ. At the end of each δ-interval, all the environment variables are evaluated in parallel, and then the control outputs serially as per the topological sort of the blocks. The Euler method of integration is used for updating the environment variables in steps of an integration step δ. In all the other cases below, the environment variables are evaluated in the same way.
- *Discrete-time semantics* (`disc`): This is same as the `cont` case except that the control variables are updated only in intervals of the parameter Δ. The simulator uses $\Delta = 1$.
- *Round-robin dispatch sequence* (`rr`): The blocks are executed as per the dispatch sequence σ^{RR}. The order of blocks in σ^{RR} is given by the topological sort of B_G. The execution time of B_i is chosen uniformly at random between $\gamma_l(B_i)$ and $\gamma_u(B_i)$ for each instance of B_i. The block B_i samples the values of Y_i when it starts execution, and the variables in U_i are updated at the end of the execution.
- *NPEDF dispatch sequence* (`npedf_ds`): The execution of this is the same as that of `rr` except that the dispatch sequence used is σ^{NPEDF}.

- *EDF dispatch sequence* (`edf_ds`): This case is interesting because of the need to simulate splitting of block-codes. Let the split-blocks produced for B_i be B_{ij} (after optimization). Let the relative execution time of B_{ij} be τ_{ij}^r. Assuming that the relative execution time τ_i^r corresponds to an actual execution time of $\gamma_u(B_i)$, the execution time of B_{ij} is $(\tau_{ij}^r \times \gamma_u(B_i))/\tau_i^r$. Now, for any particular execution of B_i, the execution time τ_i may be less than $\gamma_u(B_i)$. In such a case, we execute the blocks in the order $B_{i1}, B_{i2}, \ldots$ until an execution time of τ_i is consumed, and the remaining blocks are not executed.
- *NPEDF schedule* (`npedf_sch`) and *EDF schedule* (`edf_sch`): These are simulated using the NPEDF and EDF scheduling algorithms. A min-priority queue is used to extract the block with the earliest deadline.

The outputs of the simulator are the following:

- For each variable v, the value of v after each δ-interval from time $t = 0$ to time $t = N$.
- The value of a measure *opt* for each strategy. This measure is used for assessing the performance of the strategies. It is application specific, and is calculated as a function of the plant variables during the course of simulation.

5.2. Robot navigation example

The performance measure *opt* in this case is the total distance D traveled by the robot from the source S to the target T. The simulation parameters are $N = 500$, $\Delta = 0.1$, $S = (0, 0)$, $T = (200, 200)$, ROBOT_SPEED $= 2.0$, MINRAD $= 10.0$, $O_0 = (90, 110)$, $O_1 = (260, 50)$, and $O_2 = (50, 260)$. The relative execution times and relative periods are those in Fig. 5. The concrete periods used are those in Fig. 2. The simulation results for three sets of simulations using the above parameters for different γ_l and γ_u are shown in Fig. 6 for all 7 strategies. The notation used is $\gamma(B_i) = [\gamma_l(B_i), \gamma_u(B_i)]$. The execution times for I and II are taken from Fig. 2 and in both cases, $\gamma_l(B_i) = \gamma_u(B_i)$. The τ_i^r's are scaled by 3 for I, and by 6 for II. For III, the τ_i^r's are scaled *roughly* by 6 so that $\gamma_l(B_i) < \gamma_u(B_i)$.

The lower the value of D, the better the strategy is. It can be seen, as expected, that `cont` and `disc` always perform much better than the other strategies. In all cases (except that of `edf_ds` of I), round-robin performs worse than the other dispatch-sequence generation strategies. These strategies ensure computation of θ in between the estimation of obstacle radii, and this helps the robot to take advantage of recently computed obstacle radii to change its course. This also demonstrates that taking into account the relative periods of the tasks can improve control performance. Next, observe that in I, the dispatch-sequence NPEDF strategy performs better than the concrete NPEDF scheduling strategy because the latter has a lot of idle times, while the former has none. In III, the dispatch-sequence generation strategies perform better than the schedule-based ones. This is because while the former schedule the next block immediately after the current block finishes execution, concrete scheduling strategies have to wait for the task to be released at the beginning of its period. Thus, the former can take advantage of tasks finishing earlier than their worst possible execution times.

5.3. Heater example

Ivancic and Fehnker [8] discuss benchmarks for verification of hybrid systems, and this example is adapted from one of their benchmarks.

	I $\gamma(B_{\{0,1,2\}}) = [12, 12]$ $\gamma(B_3) = [3, 3]$	II $\gamma(B_{\{0,1,2\}}) = [24, 24]$ $\gamma(B_3) = [6, 6]$	III $\gamma(B_{\{0,1,2\}}) = [21, 24]$ $\gamma(B_3) = [4, 6]$
	D	D	D
`cont`	346.52	346.52	346.52
`disc`	348.42	348.42	348.42
`rr`	599.08	967.34	914.78
`npedf_ds`	499.10	428.88	419.46
`npedf_sch`	575.44	428.90	518.54
`edf_ds`	605.80	518.72	512.06
`edf_sch`	579.38	649.50	560.86

Fig. 6. Simulation results for robot navigation example.

The benchmark deals with a set of rooms in a house being heated by a limited number of heaters and sharing the heaters so as to maintain some minimum temperature in all the rooms. The number of heaters is strictly less than the number of rooms. The temperature x_i of a room R_i depends linearly on the temperatures of the adjacent rooms, on the outside environment temperature u, and on the discrete variable h_i, which is 1 if a heater is present in the room and switched on, and 0 otherwise. The equation governing the rate of change of x_i is

$$\dot{x}_i = c_i h_i + b_i(u - x_i) + \sum_{i \neq j} a_{i,j}(x_j - x_i),$$

where $a_{i,j}$, b_i, c_i are constants. Each room R_i has two thresholds on_i and off_i such that the heater, if present in the room, is switched on if x_i is below on_i and switched off if x_i exceeds off_i. Each room may have at most one heater. If R_i does not have one, a heater can be fetched from an adjacent room R_j provided R_j has a heater, x_i is below a certain threshold get_i and $x_j - x_i \geqslant diff_i$. If there are more than one such rooms R_j, the strategy can choose non-deterministically to get a heater from any of those rooms.

Our example has 3 rooms, R_0, R_1 and R_2, where R_1 is adjacent to R_0 and R_2, and R_0 and R_2 are not adjacent. There are two heaters, initially switched on and in R_0 and R_1. The outside temperature is constant at $u = 4$, and $x_i = 20$ initially for all i. The thresholds are the same for all the rooms and are $off = 21$, $on = 20$, $get = 18$, and $diff = 1$.

The environment variables are x_i and the differential equations governing behavior of x_i are given by

$$\dot{x}_1 = -0.9x_1 + 0.5x_2 + 0.4u + 6h_1,$$
$$\dot{x}_2 = 0.5x_1 - 1.3x_2 + 0.5x_3 + 0.3u + 7h_2,$$
$$\dot{x}_3 = 0.5x_2 - 0.9x_3 + 0.4u + 8h_3.$$

The controller has two blocks, B_0 for shifting heaters from one room to another if necessary and B_1 for switching on or switching off all the heaters. There are six boolean control variables: hp_0, hp_1 and hp_2 indicating the presence of heaters in the rooms, and hs_0, hs_1 and hs_2 such that hs_i is 1 iff there is a heater in the room R_i and is switched on. The block diagram of the model is shown in Fig. 7.

We measured the *minimum temperature* η reached in any of the rooms during the simulation, and the *total duration* ξ for which the temperature in one of the rooms was below a certain threshold temperature $x_{\min}$. The simulation parameters are $N = 100$, $\Delta = 0.01$, $x_{\min} = 13$, $(\tau_0^r, \tau_1^r) = (4, 1)$, $(\rho_0^r, \rho_1^r) = (4, 1)$, and $(\rho_0, \rho_1) = (24, 6)$. The value of $x_{\min}$ was chosen to be 13, slightly below the minimum temperature attained by `disc`. The relative period of B_0 is much higher than that of B_1 because we expect update of heater state in a room to be more important than shifting of heaters. However, the actual results vary a lot depending on the choice of these simulation parameters.

The simulation results for four sets of simulations using the above parameters for different γ_l and γ_u are shown in Fig. 8 for all 7 strategies. The τ_i^r's are scaled by 1 in I, and by 2 in II and for both I and II, $\gamma_l(B_i) = \gamma_u(B_i)$. In III, τ_i^r's are scaled *roughly* by 2 so that $\gamma_l(B_i) < \gamma_u(B_i)$. In IV, the values are chosen such that the task set was not schedulable using the platform-dependent NPEDF strategy.

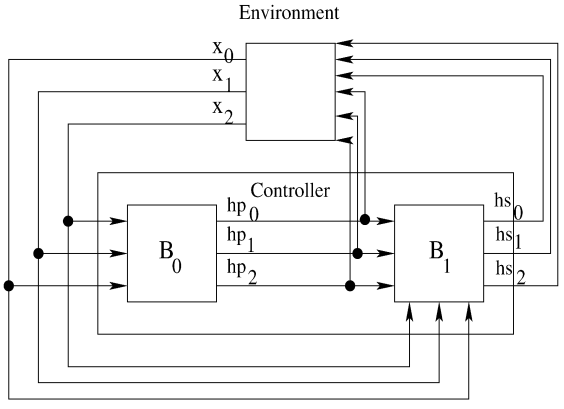

Fig. 7. Block diagram of heater model.

	I $\gamma(B_0) = [4, 4]$ $\gamma(B_1) = [1, 1]$		II $\gamma(B_0) = [8, 8]$ $\gamma(B_1) = [2, 2]$		III $\gamma(B_0) = [6, 8]$ $\gamma(B_1) = [1, 2]$		IV $\gamma(B_0) = [9, 9]$ $\gamma(B_1) = [3, 3]$	
	η	ξ	η	ξ	η	ξ	η	ξ
`cont`	15.74	0	15.74	0	15.74	0	15.74	0
`disc`	13.31	0	13.31	0	13.31	0	13.31	0
`rr`	7.79	61.01	6.88	67.78	6.91	58.71	6.58	76.24
`npedf_ds`	11.20	51.59	9.64	48.7	10.88	41.72	7.14	59.44
`npedf_sch`	9.90	50.01	9.64	50.67	9.69	49.08	–	–
`edf_ds`	10.99	48.31	9.61	40.71	10.43	43.94	8.66	44.04
`edf_sch`	9.90	50.01	7.35	59.99	8.68	57.31	8.66	52.36

Fig. 8. Simulation results for heater example.

Now, the higher the η value, and the lower the ξ value, the better the strategy is. It can be seen that the performance of `rr` is worse than that of `npedf_ds` and `edf_ds` as in the navigation example. Next, observe that if $\gamma_l(B_i) < \gamma_u(B_i)$, we expect the concrete scheduling strategy to perform worse than the corresponding dispatch-sequence strategy because the former always assumes that B_i takes $\gamma_u(B_i)$ time. This can be seen from the results in III. Next, in IV, while the tasks are not schedulable using NPEDF-scheduling strategy, the NPEDF dispatch sequence performs quite well, that is, much better than round-robin.

6. Discussion and conclusions

We have proposed an approach to generate a dispatch sequence, instead of a schedule based on real-time tasks with deadlines and periods, from a set of interacting control blocks. This proposal is relevant when there are no hard real-time deadlines, or when the implementation platform does not offer support for real-time tasks. The generation strategy itself uses relative measures inspired by scheduling algorithms, and our simulation experiments suggest that it outperforms naive methods such as round-robin in optimizing application-level performance metrics.

There are many directions for future work. Extensive experimental validation and fine tuning of the proposed approach will be necessary. In particular, we are integrating the dispatch-sequence generation strategy in the system ROCI developed for robotics applications [6]. In our examples, the dispatch sequence is supposed to imitate the timed model as best as one can, and there are no hard real-time requirements. However, a more general framework would integrate dispatch-sequence generation with application-level real-time constraints. The current generation strategy does not take into account the interdependence among control blocks due to their inputs and outputs. Also, we have assumed that there is a single processor dedicated to the controller, and this can be relaxed. Finally, it is worth exploring if control design and dispatch-sequence generation can be integrated so that some optimality guarantees of performance of the generated dispatch sequence can be obtained. Some progress towards this goal in the context of time-triggered platforms is recently reported in [22].

References

[1] R. Alur, F. Ivancic, J. Kim, I. Lee, O. Sokolsky, Generating embedded software from hierarchical hybrid models, in: Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems, 2003, pp. 171–182.
[2] K. Aström, B. Wittenmark, Computer-Controlled Systems: Theory and Design, Prentice Hall, 1997.
[3] G. Berry, G. Gonthier, The synchronous programming language ESTEREL: Design, semantics, implementation, Technical Report 842, INRIA, 1988.
[4] G. Buttazo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Kluwer Academic, 1997.
[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, Translating discrete-time Simulink to Lustre, in: Proceedings of Third International Conference on Embedded Software, in: Lecture Notes in Comput. Sci., vol. 2855, 2003, pp. 84–99.
[6] L. Chaimowicz, A. Cowley, V. Sabella, C. Taylor, ROCI: A distributed framework for multi-robot perception and control, in: Proc. IEEE Intl. Conf. on Intelligent Robots and Systems, 2003, pp. 266–271.
[7] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Luvig, S. Neuendorffer, S. Sachs, Y. Xiong, Taming heterogeneity—The Ptolemy approach, Proc. IEEE 91 (1) (2003) 127–144.
[8] A. Fehnker, F. Ivancic, Benchmarks for hybrid systems verification, in: Hybrid Systems: Computation and Control, Proceedings of the 7th International Workshop, in: Lecture Notes in Comput. Sci., vol. 2993, Springer-Verlag, 2004, pp. 326–341.
[9] N. Halbwachs, Synchronous Programming of Reactive Systems, Kluwer Academic, 1993.

[10] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous dataflow programming language Lustre, Proc. IEEE 79 (1991) 1305–1320.
[11] R. Heckmann, M. Langenbach, S. Thesing, R. Wilhelm, The influence of processor architecture on the design and the results of WCET tools, Proc. IEEE 91 (7) (2003) 1038–1054.
[12] T. Henzinger, B. Horowitz, C. Kirsch, Giotto: A time-triggered language for embedded programming, Proc. IEEE 91 (1) (2003) 84–99.
[13] T. Henzinger, C. Kirsch, The embedded machine: Predictable, portable, real-time code, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2002, pp. 315–326.
[14] Y. Hur, J. Kim, I. Lee, J. Choi, Sound code generation from communicating hybrid models, in: Hybrid Systems: Computation and Control, Proceedings of the 7th International Workshop, in: Lecture Notes in Comput. Sci., vol. 2993, 2004, pp. 432–447.
[15] K. Jeffay, D.F. Stanat, C.U. Martel, On non-preemptive scheduling of periodic and sporadic tasks, in: Proceedings of the IEEE Real-Time Systems Symposium, 1991, pp. 129–139.
[16] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, Model-integrated development of embedded software, Proc. IEEE 91 (1) (2003) 145–164.
[17] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, Kluwer Academic, 2000.
[18] E. Lee, What's ahead for embedded software, IEEE Comput. 33 (9) (September 2000) 18–26.
[19] S. Sastry, J. Sztipanovits, R. Bajcsy, H. Gill, Modeling and design of embedded software, Proc. IEEE 91 (1) (2003) 3–10.
[20] D. Seto, J. Lehoczky, L. Sha, K. Shin, On task schedulability in real-time control systems, in: Proceedings of the IEEE Real-Time Systems Symposium, 1996.
[21] M.D. Wulf, L. Doyen, J. Raskin, Almost ASAP semantics: From timed models to timed implementations, in: Hybrid Systems: Computation and Control, Proceedings of the 7th International Workshop, in: Lecture Notes in Comput. Sci., vol. 2993, 2004, pp. 296–310.
[22] H. Yazarel, A. Girard, G.J. Pappas, R. Alur, Quantifying the gap between embedded control models and time-triggered implementations, in: Proceedings of the IEEE Real-Time Systems Symposium, 2005.